Systematically Refactoring Inheritance to Delegation in JAVA

Hannes Kegel
ej-technologies GmbH
Claude-Lorrain-Straße 7
D-81543 München
hannes.kegel@ej-technologies.com

Friedrich Steimann
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen
steimann@acm.org

Abstract

Because of the strong coupling of classes and the proliferation of unneeded class members induced by inheritance, the suggestion to use composition and delegation instead has become commonplace. The presentation of a corresponding refactoring in the literature may lead one to believe that such a transformation is a straightforward undertaking. However, closer analysis reveals that this refactoring is neither always possible, nor does it necessarily achieve its desired effect. We have therefore identified the necessary preconditions and realizable postconditions of the refactoring, and built a tool that can perform it completely automatically. By applying this tool to all subclasses of several open-source projects, we have collected evidence of the applicability of the refactoring and of its capability to deliver on its promises. The refactoring builds on constraint graphs originally developed for type inference to check the preconditions and to compute the necessary delegation as well as the subtype relationships that must be maintained.

Categories and Subject Descriptors D.2.3 [**Software Engineering**]: Coding Tools and Techniques – object-oriented programming. D.3.3 [**Programming Languages**]: Language Constructs and Features – inheritance, patterns, polymorphism.

General Terms Design, Experimentation, Languages.

1. Introduction

Favor object composition over class inheritance.

Second principle of object-oriented design from [7]

A subclass uses only part of a superclasses interface or does not want to inherit data.

Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing.

Synopsis of the REPLACE INHERITANCE WITH DELEGATION refactoring from [5]

Inheritance in object-oriented programs is both a blessing and a curse. It is a blessing because it allows reuse of implementation with minimal effort; it is a curse because it establishes a strong coupling between classes and because it tends to bloat the interfaces of subclasses with unneeded members. The latter is particularly a problem in languages like JAVA, whose notion of subclassing mixes the concepts of inheritance and subtyping so that the former cannot be enjoyed without the latter.

One way out of this dilemma is to replace inheritance with delegation [5, 7, 8, 10, 14, 15, 23]. This delegation, which builds on object composition, is captured by the so-called DELEGATION PATTERN [10] according to which an object receiving requests, the *delegator*, passes them on to another, the *delegatee*[1], which it privately owns and which does the actual work. In programs using inheritance, this pattern is introduced through a corresponding refactoring prescribing all necessary checks and changes. Such a refactoring, named REPLACE INHERITANCE WITH DELEGATION by Fowler [5], has been described by several authors in the literature (e.g., [5, 8, 15]); however, the descriptions remain rather cursory.

Experience has taught that the systematic application of informally described refactorings to concrete programs brings a number of problems and pitfalls to light. In fact, much like for many other type-related refactorings [31], it turns out that for REPLACE INHERITANCE WITH DELEGATION the devil is in the details. It is therefore highly desirable that the refactoring be formalized by giving its exact preconditions and postconditions, and that the necessary program transformations be automated by a corresponding refactoring tool.

This paper reports on the results of such an endeavour. More specifically:

- We perform an informal analysis of the REPLACE INHERITANCE WITH DELEGATION refactoring as described by Fowler and others and identify a list of preconditions and postconditions of its application (Section 3).
- We present the implementation of a corresponding refactoring tool that builds on type constraints to determine the amount of delegation and subtyping needed (Section 4).
- We demonstrate the viability of this refactoring by presenting concrete numbers on the frequency of its applicability and the effect of its use in several sample projects (Section 5).

A discussion of the results and a comparison with related work conclude our paper.

2. Why to replace inheritance with delegation

Inheritance is a wonderful thing, but sometimes it isn't what you want. Often you start inheriting from a class but then find that many of the superclass operations aren't really true of the subclass. In this case you have an interface that's not a true reflection of what the class does. Or you may find that

ICSE'08, May 10–18, 2008, Leipzig, Germany.

[1] We use *delegatee* rather than the more common *delegate* to refer to the object to which is delegated, since the word *delegate* has a different, misleading connotation in everyday language.

```
class StackUser {
  Stack s = new Stack();
    …
    s.push(…);
    if (s.size() …
}

// class Stack before the refactoring
public class Stack extends Vector {
  public Stack() {}
  public Object push(Object item) {
    addElement(item);
    return item;
  }
}

// class Stack after the refactoring
public class Stack {
  protected Vector delegatee;
  public Stack() {
    delegatee = new Vector();
  }
  public Object push(Object item) {
    delegatee.addElement(item);
    return item;
  }
  public int size() {
    return delegatee.size();
  }
}
```

Figure 1. A stack user and two alternative stack implementations, one using inheritance, the other delegation. See also Figure 2.

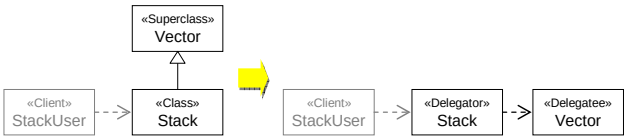

Figure 2. Dependencies and subclassing before and after the refactoring (UML notation). Note that the roles of the classes have changed: `Stack` has changed role from *Class* to *Delegator*, and `Vector` has changed role from *Superclass* to *Delegatee*.

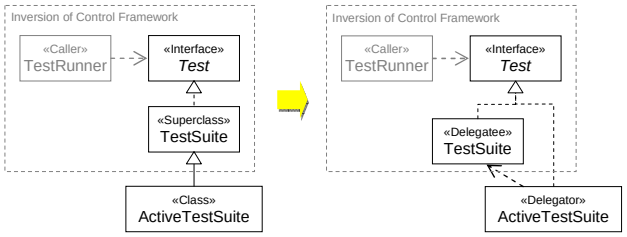

Figure 3. Replacing inheritance with delegation where subtyping is required (example taken from JUNIT 3.8 [16]).We will see in Section 3 that this refactoring breaks the code. Can you tell why?

> *you are inheriting a whole load of data that is not appropriate for the subclass. Or you may find that there are protected superclass methods that don't make much sense with the subclass.*
>
> *You can live with the situation and use convention to say that although it is a subclass, it's using only part of the superclass function. But that results in code that says one thing when your intention is something else — a confusion you should remove.*
>
> *By using delegation instead, you make it clear that you are making only partial use of the delegated class. You control which aspects of the interface to take and which to ignore. The cost is extra delegating methods that are boring to write but are too simple to go wrong.* [5, p. 352]

Everyone who has used a contemporary IDE and who has extended an existing application framework such as AWT with it has experienced the problem of bloated class interfaces: automatic code completion offers literally hundreds of members for an inheriting class to choose from, even though only a small fraction of these will actually be used. Beyond mere nuisance, this can become a real problem if inherited methods break the contract of a class. The prototypical example of this is the class `Stack` from the `java.util` library, which inherits from `Vector` and its superclass `AbstractList` 45 methods, many of which (such as `insertElementAt(.,.)`) are inappropriate for stacks. In presence of subtyping (which is tied to inheritance in JAVA) the problem is worsened by the fact that contract violations need not appear directly in the code: since a `Stack` object can be assigned to a `Vector` variable, violation can also occur through an unsuspiciously typed alias. These problems are easily solved by *replacing inheritance with delegation* [5] (sometimes also referred to as *replacing inheritance with composition* [8] or *aggregation* [23]), i.e., by dropping the inheritance relationship (and with it the inappropriate subtyping), by composing the `Stack` object of an instance of `Vector` instead, and by forwarding all legal requests from the stack to the vector. Figures 1 and 2 show the situation before and after the refactoring; note that access from clients (including subclasses) to inherited, but not overridden members (such as access to `size()` from `StackUser`) requires additional delegation, as do calls on `super` from within `Stack` (not shown).

Quite obviously, the refactoring cannot always be applied. For instance, if *Superclass*[2] is abstract, it will not work since abstract classes cannot be instantiated so that no delegatees can be created. On the other hand, a primary purpose of abstract classes (at least in JAVA, which has interfaces as an alternative supertype construct) is to let other classes inherit their (incomplete) implementations, so that removing inheritance should not be an issue in these cases. However, as we will see below there are other obstacles to replacing inheritance with delegation.

Besides removing inappropriate subtyping and with it the bloating of class interfaces, the refactored code has other advantages when compared to the original version. One is that delegation preserves encapsulation of *Delegator* and *Delegatee*, whereas inheritance breaks that of *Superclass* and *Class* [23] (the so-called *inheritance* [11] or *specialization interface* [26], whose implicitness is at the heart of the *fragile base class problem* [22]; see also Section 3.1.2). Another is that delegation allows dynamic replacement of the delegatee [20], which cannot be done with the superclass in most object-oriented programming languages. Last but not least, replacing an existing inheritance relationship with delegation allows the introduction of a new one, and is therefore an effective means of emulating multiple inheritance in single inheritance languages.

However, inheritance is not only used for code reuse as in the `Stack` example, it is also a means of extending white-box frameworks into applications [4, 7, 14]. In these cases, subtyping is mandatory to let the subclasses enjoy the functionality provided by the framework. For instance, replacing inheritance of the class `ActiveTestSuite` from class `TestSuite` in the JUNIT unit testing framework [16] with delegation breaks the code, since due to the lost subtype relationship `ActiveTestSuite` can no longer be plugged into the framework (so that methods of `ActiveTestSuite` can-

[2] In the following, *Class* and *Superclass* refer to the roles the classes play before the refactoring, and *Delegator* and *Delegatee* to the ones after (cf. Figures 2 and 3).

not be invoked through the framework; a so-called *inversion of control* [4]). However, inspection of the JUNIT source code reveals that it is sufficient for ActiveTestSuite to subtype the Test interface (which is a supertype of TestSuite), so that the refactoring shown in Figure 3 produces no typing errors. As we will learn in Section 3.1.2, it introduces another error, though.

It is not the task of the refactoring to decide whether the subtyping that goes along with inheritance is desired (as in the ActiveTestSuite example) or a flaw (as in the Stack example). Instead, the refactoring must try to achieve its purpose, replacement of inheritance, while at the same time make sure that the refactored program is still type correct and that it behaves the same. The above examples already suggested that to ensure this, performing the refactoring requires a prior program analysis to decide which subtyping relationships can be abandoned from a program without breaking it. However, this is not the only problem; in fact, the correct specification of the refactoring is all but trivial.

3. Specification of the refactoring

In the above examples, the changes required by the REPLACE INHERITANCE WITH DELEGATION refactoring are limited to the class for which inheritance is to be replaced. We believe this to be essential for the adoption of the refactoring (if only because one does not always have access to all clients of a class), and pose it as a postcondition[3] of it. Another postcondition is that the inheritance is removed. But no gain without pain: as the above examples have suggested, application of the refactoring also has some hard preconditions. This requires a careful analysis.

3.1 Analysis

Since one goal of the refactoring is to reduce the size of the protocol of *Class* (the class to be refactored), an immediate question is which methods *Delegator* (the refactored class) must maintain. This set of methods can be derived from the program by means of a type inference analogous to the one described in [17, 27], i.e., by computing the smallest interface of *Class* that can replace it in all variable declarations. The methods contained in this interface are the ones for which delegating methods must be introduced. However, *Class* may have subclasses, in which case their needs (as expressed by their clients and by their own reliance on inherited methods) must also be taken into consideration when determining the needed delegation. Furthermore, use does not decide alone over which methods the refactored class must possess — the subtyping present in the program also poses its requirements.

3.1.1 Subtyping

As noted in Section 2, removing inheritance breaks the supertype chain, but subtyping may be needed to maintain type correctness. In particular, if an assignment of an instance of *Class* to a variable of type *Superclass* (or a corresponding method return) exists, the refactoring is impossible, since the necessary assignment compatibility (subtyping!) is removed by the refactoring. The same is true if an upcast to *Superclass* or a corresponding type test (such as instanceof) exists in the program. Also, there can be type constraints derived from generic types that have equivalent effect ([17]; see Section 4.1).

Things are different if only assignment compatibility to supertypes of *Superclass* is required: as suggested by the example of ActiveTestSuite (Figure 3), this can be restored by letting *Class* subtype them directly. However, if these types are interfaces, *Class* must deliver implementations for their methods, independently of whether they are actually needed by its clients. If one of these types is a class, *Class* must subclass this class (thus inheriting its methods) and deliver implementations for all abstract methods, again independent of any actual need. It follows that the subtyping required by the program can impose methods on the refactored class unneeded by its clients (so-called *dead methods*).[4]

```
public class TestSuite implements Test {
  …
  public void run(TestResult result) {
    for (Test test : fTests) {
      …
      runTest(test, result);
    }
  }
  public void runTest(Test test, TestResult result) {
    test.run(result);
  }
  …
}

public class ActiveTestSuite extends TestSuite {
  …
  public void run(TestResult result) {
    …
    super.run(result);
    …
  }
  public void runTest(final Test test,
         final TestResult result) {
      …
      test.run(result);
      …
  }
}
```

Figure 4. Excerpt from the JUNIT 3.8 unit testing framework.

The case in which *Class* must subclass a superclass of *Superclass* requires some extra attention. It means that *Class* inherits members that are also inherited by *Superclass*. While this is no problem for behaviour if all methods required from *Class* formerly inherited from *Superclass* are overridden with delegating methods, it could be a problem for state: the fields defined in the superclass are now available in both *Class* and *Superclass*. The refactoring must therefore make sure that the methods defined in *Class* do not depend on fields inherited from its new superclass. As we will see below (Section 3.1.3), certain preconditions of the refactoring ensure that this is automatically the case, unless dependence on state is implicit. This however is only the case for certain built-in language constructs, such as synchronization.

3.1.2 Forwarding vs. delegation

A characteristic property of object-oriented programming is that the clients of a class are not the only ones accessing it: its superclasses and subclasses also do, via the *inheritance* or *specialization interface* [11, 26]. This interface has two sides: the subclass's side, defining which members of the superclass are being "imported" via inheritance, and the superclass's side, defining which members of the subclass the superclass accesses via the late binding of methods called on this. This latter phenomenon, which is often overlooked, is sometimes referred to as *open recursion* [25]: it is the basis of many important design patterns (most prominently, TEMPLATE METHOD [7]), but also at the heart of the fragile

[3] In the literature, refactorings are often specified using preconditions alone. We add the postconditions here in order to specify the exact behaviour of the refactoring.

[4] If *Class* declares to implement interfaces directly, this is not changed by the refactoring: removal of what is sometimes called *interface inheritance* is no goal (because there is nothing inherited, only subtyping, which is likely present for good reasons). This means that *Class* must continue to deliver implementations for the methods required by its interfaces, even if they are actually dead. Cf. Section 4.1.3.

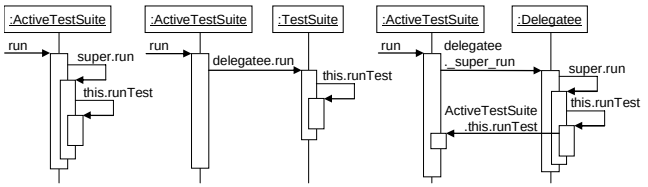

Figure 5. Left: before the refactoring. Middle: changed semantics after the refactoring; runTest in ActiveTestSuite is never called. Right: reverse delegation restoring original semantics.

base class problem [22]. When replacing inheritance with delegation, care must be taken that both sides of the interface are considered.

The code fragment from JUNIT 3 [16] shown in Figure 4 illustrates the problem. When the framework calls run on an instance of ActiveTestSuite, it calls, via super, run in TestSuite. This in turn calls runTest, but because the receiver, this, is an instance of ActiveTestSuite, the overridden version in ActiveTestSuite is invoked (Figure 5, left). Once the inheritance is removed, however, this in TestSuite points to a different object (the delegatee) than this in ActiveTestSuite (pointing to the delegator) so that run invokes runTest in TestSuite rather than in ActiveTestSuite (Figure 5, middle). And indeed, after removing inheritance from ActiveTestSuite and replacing it with delegation as in the Stack example of Figure 2, JUNIT no longer passes its own unit tests.

In the context of languages in which delegation completely replaces inheritance (such as SELF [32]), a careful distinction is made between *delegation* and *forwarding*: under delegation, this always points to the delegator, while under forwarding, this always points to the receiver of a method [20] (Figure 6). To achieve delegation in languages without a native implementation of the concept (such as JAVA), a *reverse delegation* (or, rather, a *reverse forwarding*) must be introduced: methods of *Class* formerly called from *Superclass* via open recursion must now be explicitly dispatched to *Delegator*. However, this would either require changing the implementation of *Superclass* to hold a reference to *Delegator*, which is not desired (if only because the changed implementation would affect and get inherited by other subclasses of *Superclass* for which inheritance is not to be replaced, so that open recursion must remain intact), or an interception of the problematic method calls. Fortunately, the latter is possible in JAVA via a simple trick.

The trick is to have an inner class of *Delegator* subclass *Superclass*, and to add reversely forwarding methods for all methods formerly overridden in *Class* or any of its subclasses to this new class (Figure 7). Since in JAVA an instance of a non-static inner class is always tied to an instance of its outer class (the so-called *enclosing instance* [9]), and because this instance can be accessed from the enclosed instance via <OuterClass>.this, reverse forwarding requires no extra fields or initialization. (Note that the reversely forwarding calls are dynamically bound, so that methods overridden in subclasses of *Class* are also reached.) Also, because the inner class inherits from *Superclass*, it can serve as *Delegatee* in much the same way as *Superclass* does for the forwarding sketched in Figure 2. The only caution that needs to be taken is that constructors of *Delegatee* do not call methods of *Delegator* (via reverse delegation) that themselves delegate to the delegatee (since the delegatee is not yet defined), and that a method call on super in *Class* must not be refactored to a call on the delegatee in *Delegator* if a corresponding reverse delegation exists in *Delegatee*, because this would block execution of the method in *Superclass* (the target of the initial call) and might even lead to infinite

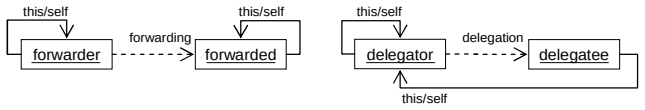

Figure 6. Difference between forwarding and delegation as implemented in prototype-based languages (UML object diagram).

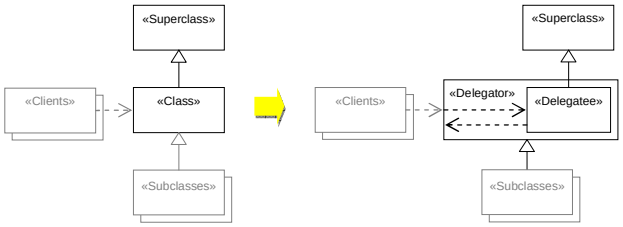

Figure 7. Refactoring using an inner subclass and reverse delegation to achieve true delegation and enable open recursion.

recursion. In case such calls to super occur in *Class*, *Delegatee* must have additional methods delegating to super, which are then called from *Delegator* (Figure 5, right; the same is necessary for methods reversely delegated to *Delegator* because one of *Class*'s subclasses, but not *Class* itself, overrode it, so that reverse delegation calls the original method in *Superclass* if invoked on an instance of *Class*). Figure 8 shows the result of this refactoring applied to ActiveTestSuite; note that an infinite recursion would occur had the call to super not been treated specially.

Not surprisingly, the price of the increased applicability of the refactoring is a loss of some of its promised gains: because inheritance is not removed, only hidden, encapsulation of *Superclass* is still broken, the fragile base class problem persists, and the possibilities to exchange the class of the delegatee at runtime are greatly reduced (all candidates must be inner classes of *Delegator*; but see the discussion in Section 6). Given these drawbacks, the refactoring should always try to use forwarding if possible (in

```
public class ActiveTestSuite implements Test {
  private class Delegatee extends TestSuite {
    …
    public void run(TestResult result) {
      ActiveTestSuite.this.run(result);
    }
    public void runTest(Test test, TestResult result) {
      ActiveTestSuite.this.runTest(test, result);
    }
    public void _super_run(TestResult result) {
      super.run(result);
    }
    …
  }
  private final Delegatee delegatee;
  …
  public ActiveTestSuite() {
    delegatee = new Delegatee();
  }
  public void run(TestResult result) {
    …
    delegatee._super_run(result);
    …
  }
  public void runTest(final Test test,
                      final TestResult result) {
    …
  }
  …
}
```

Figure 8. Refactored version of Figure 4 with reverse delegation from an inner subclass and delegation to super.

fact, forwarding is what is suggested by the descriptions of the refactoring found in the literature, so that it should have really been named REPLACE INHERITANCE WITH FORWARDING); only if not it should resort to (true) delegation. Fortunately, forwarding seems to be sufficient in most practical cases; at least this is what is suggested by the results of our experiments presented in Section 5.1.

3.1.3 JAVA specific issues

There are also several JAVA specific issues that limit the applicability of the refactoring. Among these are:

Fields In JAVA, the fields of a class can be directly accessible to its clients and its subclasses. This accessibility extends to inherited fields. If inheritance is removed, the inherited fields vanish from the interface of the class. Because JAVA cannot wrap field access via properties (as, e.g., C# or EIFFEL can), there is no way of redirecting field access to the delegatee without changing the clients. It follows that if access to the fields of a class that are not declared in the class itself is required, the refactoring is not applicable. Note that applying the ENCAPSULATE FIELD refactoring [5] first can remove this impediment (but see Section 5.1).

Non-public members Once inheritance is removed, members of a superclass that are declared `protected` become invisible if the superclass is not in the same package as the inheriting class. It follows that access to protected members from different packages prevents simple forwarding as for `Stack` in the example above (Figure 1). However, if delegation as suggested by Figure 7 is used instead, *Delegatee* (which inherits from *Superclass*) can provide access to all inherited methods, by providing a public method delegating to `super`.

Static members In JAVA, static members are also inherited and, depending on their access modifiers, accessible by the clients of a class. Again, removal of inheritance means that prior direct access to inherited members is lost. However, calls of static methods can be forwarded to the former superclass. For static fields, the restrictions above apply.

Since static methods are statically bound in JAVA, they cannot be overridden (only hidden) [9]. This prevents openly recursive calls among static methods, which in turn avoids the necessity of reverse delegation. This is fortunate, since static methods in inner, non-static classes are not allowed in JAVA ([9], cf. Section 6). However, the same restrictions as for non-static members apply.

Constructors Since *Delegator* must be able to create and initialize a *Delegatee* instance, all constructors of *Delegatee* called from *Class* via `super` and also, if implicitly called before the refactoring, the default constructor, must be accessible from *Delegator*. This is only a problem for forwarding and protected constructors, since with delegation, the inner subclass gains access through inheritance (see non-public members above).

Subclassing exceptions The JAVA language specification dictates that exceptions must remain subtypes of `Throwable`, and unchecked exceptions must remain subtypes of an unchecked exception. In particular, subclasses of `Error` and `RuntimeException` must not be changed to extend `Throwable` or `Exception` directly, unless all throws of such exceptions are either caught or declared. However, replacing inheritance among exceptions with delegation does not seem useful generally, so that we exclude it from the refactoring.

Synchronization After replacing inheritance with delegation an object and its delegatee are different instances, with different monitors. This is a problem if synchronization methods such as `wait()` or `notify()` are called in both *Class* and *Superclass* methods, because `this` now points to different objects.

3.2 Specification

Based on the previous analysis, a set of preconditions can be specified that limit the general applicability of the refactoring, and decide over whether forwarding is sufficient or (true) delegation is required. If the preconditions are satisfied, the mechanics of the refactoring must change the program in such a way that the refactoring's postconditions are satisfied, the pair thus giving a complete specification of the refactoring.

3.2.1 Preconditions

It is in the nature of the refactoring that its application is limited to classes, and more specifically to direct subclasses of another class than `Object`. Also, the class to be refactored must be changeable, i.e., the source code must be under the control of the developer. Last but not least, the program using the class must be analysable, i.e., its source code must be available and the use of the class must not occur through reflection. Besides these trivial insights, the problems identified above lead to the following preconditions:

1. *Superclass* must not be abstract.
2. No type constraint requiring that *Class* be a subtype of *Superclass* must be derivable from the program.
3. For forwarding, there must be no instances of open recursion in *Superclass*. For delegation, there must be no open recursion in constructors of *Superclass* involving access to the (not yet existent) delegatee.
4. Clients of *Class* and its subclasses must not access fields inherited by *Class*. *Class* may only access inherited fields if public or located in the same package.
5. For forwarding, *Class* or its subclasses must not require access to protected members inherited from *Superclass*.
6. For forwarding, the default constructor of *Superclass* and the constructors called via `super` must be accessible from *Class* after subclassing is removed.
7. *Class* must not be a subclass of `Throwable`.
8. Synchronizing method calls must not be split among *Class* and *Superclass*.

3.2.2 Postconditions

1. *Delegator* does not extend *Delegatee* or *Superclass*.
2. *Delegator* has a new private final field that holds an instance of *Delegatee*.
3. In case of delegation, *Delegator* has a new inner class extending *Superclass* and containing the necessary reversely delegating methods and the methods forwarding to `super` as described in Section 3.1.2.
4. *Delegator* contains delegating methods for all methods called by its clients or subclasses that were formerly inherited from *Superclass* (and therefore not implemented in *Class*).
5. *Delegator*'s accesses to fields formerly inherited are qualified with delegatee field or (for static fields) its class.
6. In case of forwarding, all calls on `super` (including those to `super` in constructors) are replaced by calls on *Delegatee*; in case of delegation, those for which reverse delegation exists

```
01  public interface I {
02    void recur();
03  }
04  public class Super implements I {
05    protected float f;
06    Super() {
07      f = 1;
08    }
09    Super(float v) {
10      f = v;
11    }
12    public void recur() {
13      over();
14    }
15    void over() {
16      f = 3;
17    }
18  }
19  public class Sub extends Super {
20    Sub() {
21      super(2);
22    }
23    void over() {
24      f = 4;
25    }
26  }
27  public class Client {
28    void use() {
29      I i = new Sub();
30      i.recur();
31    }
32  }
```

Figure 9. Sample program from which the type constraints of Table 1 are generated

are replaced by calls to methods forwarding to super in *Delegatee* instead (see Postcondition 3).

7. *Delegator* extends former indirect superclass if corresponding type constraint has been derived from the program.
8. *Delegator* implements all interfaces for which corresponding type constraints have been derived from the program.
9. All other classes and types of the program have not changed.

It follows that the only members of *Delegator* other than the new delegatee field and possibly the inner subclass are those

a) required by its clients,

b) implemented in *Class* before the refactoring (including constructors),

c) inherited from another superclass (cf. Postcondition 7 above),

d) required by abstract methods of other supertypes (including interfaces; cf. Postconditions 7 and 8 above),

e) required by subclasses of *Delegator*, either directly or by way of a) or d) applied accordingly.

However, the user of the refactoring can always include additional members to be delegated, if so desired.

Note that except for numbers 3 and 6, postconditions are identical for forwarding and delegation. Also note that Postcondition 9 makes the refactoring applicable even in cases in which subclasses of *Class* cannot be altered, as is for instance the case when refactoring a framework whose client code is unavailable for change (although read access is required for the whole-program analysis).

4. Implementation of the refactoring

Logically, the refactoring falls into four parts: checking of preconditions, computation of the required forwarding methods (including reverse forwarding and forwarding to super), computation of the required subtyping, and changing the source code. Of these, the first three require a comprehensive program analysis, while the last amounts to a rather straightforward manipulation of the abstract syntax tree of the program. We will focus on the steps requiring analysis here.

Table 1. Type constraints derived from the program of Figure 9.

LINE	CONSTRAINTS
13	*This*(Super) ≤ *Decl*(Super.over)
21	*Super*(Sub) ≤ *Decl*(Super.Super)
24	*This*(Sub) ≤ *Decl*(Super.f)
29	[i] = I, [new Sub()] ≤ [i], [new Sub()] = Sub
30	[i] ≤ *Decl*(recur)

4.1 Type constraint based analysis

As it turns out, the necessary analysis can be based on type constraints derived from the declarations, assignments, and type casts found in a program, combined with checks of properties (accessibility, abstractness, etc.) of individual program elements. Type constraints have successfully been used for type-inference based solutions of various other refactoring problems (e.g., [6, 17, 18, 19, 24, 27, 30, 31]); however, the refactoring described here requires new constraints for accessing fields and non-public methods as well as for the handling of this and super.[5] Also, constraint variables have to be annotated with package and accessibility information.

To give an idea of the problems to be solved, we resort to the sample program shown in Figure 9. From this program, the type constraints of Table 1, which are restricted to the ones needed for our analysis, are derived. The refactored class Sub is shown in Figure 10.

4.1.1 Precondition checking

Checking the first two preconditions is trivial: the first is a simple lookup of the abstractness of Super, and the second amounts to searching for (a combination of) type constraints requiring that Sub be a subtype of Super. Both checks are negative in the given example.

To detect instances of open recursion (Precondition 3), an analysis similar to that sketched in [2] is necessary. For this, a new constraint is introduced which captures the methods called on this in all superclasses of Sub (here: Super.over in line 13). If such a method is overridden in Sub or any of its subclasses (here: in Sub), open recursion has been detected. Indirect open recursion, i.e., that a superclass passes this to another class which then calls late-bound methods on it (double dispatching [12]) is covered by the constraints derived from assignments of this to other variables.

Precondition 4 is checked by searching for constraints like that derived from line 24. If such a constraint has been derived from a client of Sub (a fact stored as an annotation of the constraint), or if it has been derived from Sub and the field is not declared public and Sub and Super are not in the same package, the refactoring is rejected. However, in our example this is not the case.

Since forwarding is ruled out for the refactoring (because open recursion has been detected), Preconditions 5 and 6 are not applicable. Precondition 7 is trivial again (and is checked without resorting to constraints). Precondition 8 is checked by searching for synchronized methods implemented by Sub and its subclasses. If Super or one of its superclasses also has synchronized methods, the refactoring is not performed. In the given example, this is not the case.

[5] Note that the constraints between generic types can be nontrivial, especially in presence of type bounds and wildcards [17].

```
01 public class Sub implements I {
02   private final Delegatee delegatee;
03   Sub() {
04     delegatee = new Delegatee(2);
05   }
06   void over() {
07     delegatee.f = 4;
08   }
09   public void recur() {
10     delegatee.recur();
11   }
12   private class Delegatee extends Super {
13     Delegatee(float v) {
14       super(v);
15     }
16     void over() {
17       Sub.this.over();
18     }
19   }
20 }
```

Figure 10. Changes made to Sub as a result of the refactoring.

4.1.2 Computation of required forwarding

The constraints derived from lines 29 and 30 imply that Sub ≤ *Decl*(recur) and thus that recur must be a method of Sub. Since Sub does not implement recur (it is inherited from Super), Sub must introduce a corresponding forwarding method (Figure 10, line 9). By collecting all constraints of this kind for Sub (including those in which it is represented by this or super), Sub's set of forwarding methods is computed.

The constraint derived from line 24 implies that Sub accesses f. Since f is inherited from Super, it must be replaced with delegatee.f (Figure 10, line 7).

Computation of the necessary reverse forwarding is based on the same constraints as used for the detection of open recursion: since *This*(Super) ≤ *Decl*(Super.over) and over is overridden in Sub, a reversely forwarding method is introduced in Delegatee (Figure 10, line 16). Methods and constructors forwarding to super result in constraints like that derived from line 21; it implies that Sub needs access to the constructor of Super, which is achieved by forwarding to Delegatee (Figure 10, line 4). If reverse forwarding with same signature exists in Delegatee, a new method prefixed with _super_ is introduced, to which is forwarded instead (see Section 3.1.2 for an example).

4.1.3 Computation of required subtyping

While checking Precondition 2 showed that Sub need not subtype Super, the same type constraints (here all derived from line 29) express that Sub must subtype I. A corresponding implements clause is therefore added to Sub (Figure 10, line 1). For all methods of I that are not implemented by Sub, forwarding to Delegatee is added. In the given example, this is method recur, which was already identified by the steps above.

4.2 Testing

We have tested our refactoring tool by applying it to all subclasses of various publicly available JAVA packages with good coverage by JUNIT tests. We especially looked for projects making intense use of generics, since type constraints for these are particularly difficult to get right (cf. Footnote 5). The packages, which are partly the same as those used in [19], were:

- JUNIT 3.8.1 and 4.4 (JU3 and JU4, http://junit.org)
- JAKARTA COMMONS COLLECTIONS 4.01, a popular replacement of Java's collections (JCC, http://larvalabs.com/collections/)
- JHOTDRAW 6.0 beta 1, a drawing editor framework and source of patterns (JHD, http://jhotdraw.org)
- JPAUL, a collection of algorithms widely used in program analysis (http://jpaul.sourceforge.net)

After each refactoring, we compiled the program to make sure that no typing constraints were violated, and ran all test cases to check that no behavioural changes were induced.

4.3 Verification

Because of the incompleteness of testing, we cannot be sure that the refactoring preserves semantics in all cases; for this, a formal proof using the complete language specification of JAVA would have been necessary, which exceeded our possibilities. In fact, even for restricted languages such proofs are difficult, which is why sketches are usually delivered instead (cf. Section 7). However, whenever we believed that we had made correctness of the refactoring plausible, testing it on a new project revealed a new problem we had not previously thought of. Therefore, we refrain from all proof attempts here.

4.4 Availability

We have implemented the refactoring as described here as a plug-in to the ECLIPSE IDE. It utilizes ECLIPSE's built-in refactoring framework, including full preview to all changes and undo functionality. The plug-in and a brief description of its use are available from http://www.fernuni-hagen.de/ps/prjs/RIWD.

5. Evaluation of applicability and effect

To get an impression of the practical relevance of the REPLACE INHERITANCE WITH DELEGATION refactoring, we measured its applicability (in terms of the preconditions satisfied) and its effect (in terms of the decreased size of the protocol). For this, we logged the fulfilment of preconditions and sizes of protocol before and after each refactoring, using the same packages and application procedure as for testing.

5.1 Applicability

Table 2 summarizes the applicability of the refactoring. The given reasons for non-applicability are not mutually exclusive; usually, more than one led to a rejection of the refactoring. Since delegation is always possible if forwarding is, the number of successful applications is listed under "delegation possible".

Overall, the refactoring is applicable in only 26% of all cases. However, given that Precondition 1 (a non-abstract superclass) prevents the refactoring in 55% of all cases, and that removing inheritance from abstract superclasses is usually not an issue (cf. Section 2), nor is subtyping Throwable (Precondition 7), there are only 160 subclasses for which the refactoring can be considered *relevant*. Of these, the refactoring is applicable in 63%.

Several other things can be observed from the data underlying Table 2):

- Delegation is necessary in only 37% of the cases in which the refactoring is applicable (without JHOTDRAW in only 15%). Given that forwarding is preferred (since it avoids secret subclassing; cf. Section 3.1.2), this is good news.
- Across all projects, the reason for necessity of delegation is presence of open recursion (Precondition 3) alone in 54%, access to protected members across packages (Preconditions 5 and 6) alone in 14%, and both in the rest of the cases. Note that increasing protected accessibility to public would turn the inheritance interface into a client interface, allowing forwarding (Precondition 5). If a public (client) interface is not desired, inheritance was probably the right design choice and refactoring it is obsolete.

Table 2. Applicability of the refactoring

	JU3	JU4	JCC	JPAUL	JHD	TOTAL
number of subclasses	27	50	106	31	180	394
violated preconditions						
#1: abstract superclass	10	31	65	28	82	216
#2: subtyping required	4	16	29	15	25	89
#3: open recursion	11	31	15	19	87	163
#4: access to inherited field	0	0	31	2	5	38
#5: access to protected member	6	1	21	0	34	62
#6: invisible constructor	2	2	28	4	22	58
#7: subclass of `Throwable`	2	9	3	1	3	18
#8: synchronization	3	0	1	0	10	14
forwarding possible	9	5	7	1	42	64
delegation possible	12	6	7	1	75	101
neither	15	44	99	30	105	293

- Among the reasons for non-applicability, access to inherited fields (Precondition 4) was exclusively responsible in only 6% of all relevant cases. This relatively small number justifies our decision to not encapsulate field access and change clients to use the resultant accessors as part of the refactoring (cf. Sections 3.1.3 and 7). Adding an option to the refactoring tool that would allow one to perform such encapsulation if it makes the refactoring possible might be worthwhile, though.
- Another reason that can be removed by a corresponding refactoring, the need to subtype *Superclass* (Precondition 2), exclusively accounted for 16% of all relevant rejections of application. Again, this does not justify solving the subtyping problem as part of the refactoring (but cf. Section 6).
- Last but not least, split synchronization (Precondition 8) had the least impact on applicability. This is good, since it is also the most unrelated to the purpose of the refactoring and very difficult to get around. On the other hand, it turned out to be the only reason for non-applicability of the refactoring to class `Stack` in the JDK 1.4.

5.2 Effect

Table 3 summarizes the effect of the refactoring. It lists the non-private methods of all 160 *relevant* (see above) classes, both static and non-static (a set referred to as the *protocol* of a class hereafter), before and after the refactoring, the number of delegating methods introduced, the number of types that had to be subtyped directly due to the found type constraints, and the number of methods whose implementation was required by or inherited from these supertypes, but are not otherwise needed (the *dead methods* mentioned in Section 3.1.1). Again, a few comments are in place:

- The overall reduction in size of protocol of classes is by 50%. The effect varies greatly among projects, from 31% (JCC) to 88% (JUNIT 4). Note that the large number of methods in JUNIT 3 and also JHOTDRAW comes from their GUI classes (which JUNIT 4 does not have) inheriting hundreds of methods from their base classes in SWING and AWT.
- The fact that in general only few classes had to implement additional interfaces came as a surprise to us. This may be due to a general reluctance to use interfaces (of which JHOTDRAW is a known exception) [27].
- With one exception, the case that a class had to subclass another class was limited to JUNIT 3 and JHOTDRAW, and there to GUI classes, which is explained by the fact that the GUI frameworks SWING and AWT use classes as extension points.

Table 3. Effect of the refactoring

	JU3	JU4	JCC	JPAUL	JHD	TOTAL
methods before	3205	164	181	45	11873	15468
methods after	1962	19	124	18	5633	7756
delegating methods	53	9	83	15	1072	1232
new direct interfaces	3	3	4	1	46	57
new direct superclasses	7	0	1	0	11	19
imposed dead methods	1871	1	3	12	4649	6536

- As can be seen from the number of dead methods, the subtyping constraints in a program counteract the purpose of the refactoring. Unlike with improving applicability, to increase the gain it does make sense to generalize the causing supertypes using the INFER TYPE refactoring [27] (cf. Section 6).

6. Discussion

While our evaluation has delivered concrete numbers of applicability (including individual reasons for non-applicability) and an impression of the effect achievable by the refactoring, it should be clear that our results are purely technical — in particular, no insights can be derived from them as to whether or when applying the refactoring leads to better design. On the other hand, such an effect is inherently difficult to asses, which is why we took the suggestions from some of the authorities in the field of object-oriented design referred to in the introduction for granted. However, one result of our evaluation is that inheritance has, and most likely will keep, its place in object-oriented programming.

Although the presentation in this paper is based on JAVA, many of the results can be transferred directly to other class-based, statically type-checked object-oriented programming languages, and even for untyped languages such as SMALLTALK, type inference is required to compute the necessary delegation. Some of the implementation details, such as the reliance on inner classes and enclosing instances, are specific to JAVA; however, a generalized implementation that does not rely on the availability of these concepts is possible.

Indeed, the somewhat compromised flexibility achieved with delegating to an inner class (mentioned at the end of Section 3.1.2) and the dependence on JAVA specifics can be lifted by changing the refactoring to introduce a top-level subclass of *Superclass* as *Delegatee*. Instances of this class must then have explicit links to their delegators to allow reverse delegation. If several such *Delegatee* classes are introduced and offer the same interface, a delegator can choose from these alternatives and even change its delegatee dynamically.

However, it is debatable whether the (hidden) subclassing of *Delegatee* is acceptable for the purpose of the refactoring. After all, it does not replace inheritance, but only moves it to a new class. On the other hand, it effectively reduces the size of the interface of the refactored classes, which alone is an often desired effect. Whether this is worth the more complex design must be decided by the developer.

The restriction that there must be no required assignment compatibility from *Class* to *Superclass* (Precondition 2) can be circumvented by inferring the common type of all variables to which such an assignment exists (which will be a structural supertype of *Class*), using this type in the variables' declarations, and letting both *Class* and *Superclass* subtype this type (Figure 11). Also, the

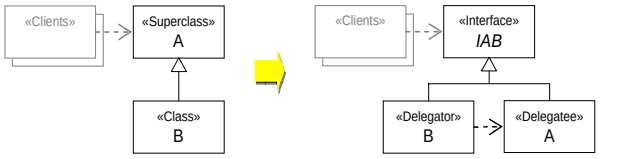

Figure 11. Making the refactoring possible in presence of type constraints requiring *Class* to subtype *Superclass*, by making clients depend on an inferred supertype instead.

bloating possibly resulting from required interface implementations (both direct and indirect) can be reduced by generalizing the implemented interfaces. However, these are different refactorings.

One useful by-product of the type inference conducted in the course of our refactoring is that it detects methods that are never called. This is because it computes two sets of members that must be offered by *Delegator*: those actually relied upon by clients or its subclasses, and those required by the typing rules (subtyping required to maintain assignment compatibility). The latter minus the former is dead code and need not have a real implementation; however, since removal of signatures may make changes in other type definitions necessary, we leave this to other refactorings (see, for instance, [29] and also the discussion in [27]). The user interface of the refactoring can give corresponding hints, though, at no extra cost.

A common concern with delegation (and with refactoring in general) is decreased performance: explicit delegation requires a method call, which especially under late binding has its price. The greatest penalty is imposed by open recursion from *Superclass* on an instance of *Class* where not *Class*, but one of its subclasses overrides the called method. In this case, the call is first reversely delegated to *Class*, then delegated back to the super-call in *Delegatee*, and from there back to *Superclass* (see Section 3.1.2), the net effect being zero. However, since *Delegatee* can be declared final, an optimizing compiler can inline two of the three delegation methods, leaving only one additional dynamic dispatch.

7. Related work

Replacing inheritance with delegation (or composition) is a recurrent theme in object-oriented programming textbooks and internet forums. Contrary to its apparent popularity, it seems that it has not been the subject of much investigation, with the notable exception of the work conducted by the pioneers of refactoring [15, 23], and by the prototype-based programming community [20, 28, 32].

Delegation as shown in Figure 6 has been recognized very early by the prototype-based programming community as a viable alternative to class-based inheritance [20]. In fact, delegation can be viewed as inheritance among individuals, which is closer to the biological concept of inheritance than inheritance among classes. SELF [32] is perhaps the best known object-oriented programming language that builds on delegation, but Sun's interest in it was superseded by that in JAVA, which promotes class-based reuse. Stein [28] has pointed out that inheritance is delegation on the class level if classes are viewed as prototypes: a member lookup that failed for a class is delegated to the superclass. However, all these insights have not helped delegation among objects become a mainstream native language construct — today, it is mostly seen as (and used in the form of) a pattern.[6]

Because of the many problems of class-based inheritance, several alternatives for reuse on the class level have been proposed. So-called mixins [1] provide an alternative to standard (multiple) inheritance by allowing the extension of different classes with the same, reusable code (the mixin). Mixins can wrap methods defined in a class by overriding them and accessing the overridden methods via super, where super refers to different classes in different uses of the mixin. Because mixins suffer from some of the same problems as subclassing [3], one could be tempted to replace them with delegation, too; as it turns out, in languages without a native mixin construct they are emulated using the DECORATOR pattern, which relies on forwarding [7].

Traits provide an alternative to mixins by providing class-like, state-less behaviour specifications (basically sets of methods) that can be used by other classes for reuse of implementation [3]. An object using a trait is basically extended by the trait's set of methods. The implementation of traits described in [3] is similar to delegation in that method calls to an object are forwarded to the trait. We conclude from this that even today, delegation appears to be the one alternative to inheritance.

In his work on typed inheritance and the inheritance interface, Hauck has shown how inheritance can be modelled as a special kind of aggregation (aka composition; cf. below) [11]. For this, he introduces two special fields, one (called super) in the subclass that points to an instance of the superclass, and one (called self) in the superclass that points to the instance of the subclass. Inherited and overridden methods are called via delegation (using super) and reverse delegation (using self), respectively; this is similar to what we are doing in presence of open recursion, except for the fact that we use an inner subclass to avoid changing the superclass. Also, this buys us implicit links (the enclosing instance) for the reverse delegation (cf. Figure 7).

Perhaps the best known reference for the REPLACE INHERITANCE WITH DELEGATION refactoring is Fowler's book [5], but as already mentioned, it only scratches the surface. Genßler and Schulz have presented a reengineering pattern that transforms inheritance to delegation (or, as they call it, composition) [8], mostly with the goal to introduce certain design patterns (namely BRIDGE, STRATEGY, and STATE from the standard design pattern catalogue [7]). However, their description of the transformation, which is kept mostly language independent, also ignores many problems: of the many practical issues we have identified above it mentions only the static type checking problems encountered when a delegator is to be used as (an instance of) its former superclass (which has already been noted in [15]); even for this, it offers no solution.

Perhaps the first (and also most analytical) treatment of the refactoring can be found in Opdyke's Ph.D. thesis [23], Chapter 8.9 (cursorily repeated in [15]). It avoids our Precondition 4 by first performing the ENCAPSULATE FIELD refactoring [5] for all inherited fields, which violates our Postcondition 9, namely that the rest of the program remains unaffected. Also, it introduces delegation in *Delegator* for *all* methods (including the new accessors) formerly inherited from *Superclass*, no matter whether they are actually needed, thereby failing to reduce the bloating of class interfaces, one of the main goals of the refactoring. Last but not least, it does not postulate absence of openly recursive calls as a precondition, thereby either accepting changed program semantics or requiring that references to delegating objects are made available (the description in [23] and also [15] is unclear in this regard).

We are aware of one other implementation of the REPLACE INHERITANCE WITH DELEGATION refactoring for JAVA, namely that deployed with INTELLIJ IDEA [13]. Like our own implementation, the tool performs a program analysis and resorts to inner classes if deemed necessary. However, the program analysis is incomplete: it ignores many assignments to superclasses contained in the pro-

[6] A delegate in C# is basically a type-safe pointer to a function bound to an object and as such a different concept than the one discussed here.

gram (violating Precondition 2 and Postcondition 7), leading to typing errors in the refactored programs; it ignores protected accessibility of members in the superclass (Postcondition 3), leading to accessing errors; etc. The refactoring itself differs in that it changes subclasses and clients of static members to reference *Delegatee* directly, thereby violating our Postcondition 9 (the refactoring even offers introduction of a getter for the delegatee so that clients of *Delegator* can address their requests directly to *Delegatee*). Also, it does not introduce reverse delegation as we do, but rather moves the methods overridden in *Class* to the inner class. This however prevents methods overridden in subclasses of *Class* from being called, which changes program semantics fundamentally.

Formal specification and correctness proofs of refactorings have been a desideratum from the beginning of the field [23]. They require a formal capture of preconditions and postconditions, as well as a proof that the postconditions follow from the preconditions given the steps of the refactoring. This is analogous to usual program verification, which is natural because refactoring tools are meta-programs. However, only few works actually provide such proofs for nontrivial refactorings: [23] provides formal preconditions (but no postconditions) and informal argumentations to make correctness plausible; [21] shows how such proofs could be conducted using graph transformations, but it is not at all clear that these proofs would be easier to arrive at than Hoare style program verification.

Recently, type inference has become the basis of a number of refactorings [31]. Starting with the type constraints described in [30] (which build on the work of [24]), several new refactorings using this technology have been developed (e.g. [6, 19]). The first author of this paper has extended the type constraint framework described in these works to cover most of the JAVA 5 language specification [17]; it has become the basis of the re-implementations of other existing refactorings such as [27], and of new ones such as INJECT DEPENDENCY and CREATE MOCK OBJECT [18].

8. Conclusion

While often advocated as a standard remedy, the replacement of inheritance with delegation is neither always possible, nor generally trivial to perform. In fact, we found that popular descriptions of the corresponding refactoring ignore many of its problems, and that in fact it is applicable to only 26% of all subclasses, and to 63% of all subclasses for which the refactoring seems relevant. Also, in 37% of all possible applications the refactored code exposes so much technical overhead that the benefit of the refactoring must be questioned. Last but not least, we found that one value of the refactoring, the reduction of the number of members of a class from what is inherited to what is actually needed, is greatly diluted by necessary subtyping: while on average, only 8% of all methods of a refactored class are actually needed by its clients or subclasses, another 42% must be maintained due to subtyping required by assignments to supertypes and other type constraints derived from the program.

References

[1] G Bracha, WR Cook "Mixin-based inheritance" in: *Proc. of OOPSLA/ECOOP* (1990) 303–311.

[2] S Demeyer "Analysis of overridden methods to infer hot spots" in: *ECOOP'98 Workshop Reader* LNCS 1543 (1998) 66–67.

[3] S Ducasse, O Nierstrasz, N Schärli, R Wuyts, AP Black "Traits: A mechanism for fine-grained reuse" *ACM Trans. Program. Lang. Syst.* 28:2 (2006) 331–388.

[4] ME Fayad, DC Schmidt "Object-oriented application frameworks" *Communications of the ACM* 40:10 (1997) 32–38.

[5] M Fowler *Refactoring: Improving the Design of Existing Code* (Addison-Wesley 1999).

[6] RM Fuhrer, F Tip, A Kiezun, J Dolby, M Keller "Efficiently refactoring Java applications to use generic libraries" in: *Proc. of ECOOP* (2005) 71–96.

[7] E Gamma, R Helm, R Johnson, J Vlissides *Design Patterns — Elements of Reusable Software* (Addison-Wesley, 1995).

[8] T Genssler, B Schulz "Transforming inheritance into composition — A reengineering pattern" in: *Pro. of 4th EuroPLoP* (1999).

[9] J Gosling, B Joy, G Steele, G Bracha *The Java Language Specification* (http://java.sun.com/docs/books/jls/).

[10] M Grand *Patterns in Java* 2nd edition (Wiley & Sons 2002).

[11] FJ Hauck "Inheritance modeled with explicit bindings: An approach to typed inheritance" in: *Proc. of OOPSLA* (1993) 231–239.

[12] DHH Ingalls "A simple technique for handling multiple polymorphism" in: *Proc. of OOPSLA* (1986) 347–349.

[13] IntelliJ IDEA (http://www.jetbrains.com).

[14] RE Johnson, B Foote "Designing reusable classes" *Journal of Object-Oriented Programming* 1:2 (1988) 22–35.

[15] RE Johnson, WF Opdyke "Refactoring and aggregation" in: *Proc. of ISOTAS* LNCS 742 (1993) 264–278.

[16] *JUnit unit testing framework* (http://junit.org).

[17] H Kegel *Constraint-basierte Typinferenz für Java 5* (Diplomarbeit, Fakultät für Mathematik und Informatik, Fernuniversität in Hagen 2007).

[18] H Kegel, F Steimann "ITcore: A type inference Package for refactoring tools" in: *Workshop on Refactoring Tools @ ECOOP* (2007).

[19] A Kiezun, MD Ernst, F Tip, RM Fuhrer "Refactoring for parameterizing Java classes" in: *Proc. of ICSE* (2007) 437–446.

[20] H Lieberman "Using prototypical objects to implement shared behavior in object-oriented systems" in: *Proc. of OOPSLA* (1986) 214–223.

[21] T Mens, N Van Eetvelde, S Demeyer, D Janssens "Formalizing refactorings with graph transformations" *J. Softw. Maint. Evol.* 17:4 (2005) 247–276.

[22] Mikhajlov, E Sekerinski "A study of the fragile base class problem" in: *Proc. of ECOOP* (1998) 355–382.

[23] W Opdyke *Refactoring Object-Oriented Frameworks* Ph.D. thesis (University of Illinois at Urbana-Champaign, 1992).

[24] J Palsberg, MI Schwartzbach "Object-oriented type inference" in: *Proc. of OOPSLA* (1991) 146–161.

[25] BC Pierce *Types and Programming Languages* (MIT Press 2002).

[26] R Stata, JV Guttag "Modular reasoning in the presence of subclassing" in: *Proc. of OOPSLA* (1995) 200–214.

[27] F Steimann "The Infer Type refactoring and its use for interface-based programming" *Journal of Object Technology* 6:2 (2007) 67–89.

[28] LA Stein "Delegation is inheritance" in: *OOPSLA* (1987) 138–146.

[29] M Streckenbach, G Snelting "Refactoring class hierarchies with KABA" in: *Proc. of OOPSLA* (2004) 315–330.

[30] F Tip, A Kiezun, D Bäumer "Refactoring for generalization using type constraints" in: *Proc. of OOPSLA* (2003) 13–26.

[31] F Tip "Refactoring using type constraints" in: *Proc. of Static Analysis, 14th International Symposium* (2007) 1–17.

[32] D Ungar, RB Smith "Self: The power of simplicity" in: *Proc. of OOPSLA* (1987) 227–242.